```
!*************************************************************************************************
!
!                                        PROJECT COMPUTER ARCHITECTUUR ARC ASSEMBLER 2004
!
!                                                *** MATRIX VERMENIGVULDIGING ***
!
!       Auteur: Pascal Vyncke
!
!*************************************************************************************************

! De matrix vermenigvuldiging is hieronder volledig uitgewerkt.  Bij elke regel staat er uitgelegd wat er zal gebeuren of waarom die regel er staat
! Verder staat alles per blok ingedeeld, telkens iets anders wat we berekenen, op die plaatsen staat met commentaar (dan niet achter de regel maar voor het blok)
! uitgelegd wat we gaan doen, hoe we het gaan doen en de principes, met soms gebruik van pseudocode om het goed te kunnen begrijpen.
! Het programma werkt volgens mij volledig foutloos, ik het het getest met verschillende soorten van matrixen (verschillende dimensies), en enkele speciale gevallen
!
!
! Het principe is de matrixvermenigvuldiging uit te voeren door middel van 3 lussen.
! De vermenigvuldiging wordt omgekeerd uitgevoerd, van rechts onderaan de matrix tot linksbovenaan.
! De reden hiervoor is dat ik eerst dacht dat het efficiënter was om een variabele te checken op het gelijk zijn aan 0, dan gelijk zijn aan een bepaald getal.
! Dit blijkt niet zo te zijn.  Maar het is EXACT hetzelfde als uit te voeren zoals we dat "manueel" zouden doen.
!
! Overzicht van de gebruikte registers:
! %r0: altijd 0, niet aan te passen
! %r1: aangeven of C en CSol overeen komen
! %r2: adres van geheugenlocatie van C
! %r3: variabele van loop1
! %r4: variabele van loop2
! %r5: variabele van loop3
! %r6: tussenwaarde, gebruikt bij checken van CSol
! %r7: -
! %r8: -
! %r9: -
! %r10: -
! %r11: tussenwaarde, gebruikt bij inladen van A[i,k]
! %r12: tussenwaarde, gebruikt bij inladen van B[k,j]
! %r13: tussenwaarde, gebruikt bij vermenigvuldigen van A[i,k] en B[k,j]
! %r14: -
! %r15: adres van pc, gebruikt bij call en jumpl
! %r16: tussenwaarde, gebruikt bij berekenen van waarde voor C[i,j]
! %r17: tussenwaarde, gebruikt bij inladen van inlezen waarde uit C[i,j]
! %r18: -
! %r19: -
! %r20: -
! %r21: -
! %r22: -
```

```
! %r23: -
! %r24: -
! %r25: waarde van dimAX
! %r26: waarde van dimAY
! %r27: waarde van dimBX
! %r28: waarde van dimBY
! %r29: antwoord van multiply (C in de voorstelling: A * B = C)
! %r30: 1e getal om met te vermenigvuldigen (A in de voorstelling: A * B = C)
! %r31: 2e getal om met te vermenigvuldigen (B in de voorstelling: A * B = C)
! --> dus in feite maar 18 registers gebruikt van de 32
!
! Ter info: van de 294 lijnen code die hier zijn, zijn er in werkelijkheid (zonder alle commentaar) 'maar' 85 lijnen assembler
!
! Lees verder alle commentaar in de code zelf, ik heb het zo goed en duidelijk mogelijk proberen uit te leggen.

        .begin                  ! Start assembling
        .org 2048               ! Start programma op adres 2048
        C .equ 4000             ! Reserveer op geheugenplaats 4000 voor C (de uitkomst)


        ! We laden de dimensies in de registers, aangezien deze veelvuldig worden gebruikt,
        ! is het sneller te werken met registers dan met load operaties uit het geheugen. Deze worden in het programma NIET gewijzigd
        ld [dimAX], %r25        ! Laad dimAX in %r25
        ld [dimAY], %r26        ! Laad dimAY in %r26
        ld [dimBX], %r27        ! Laad dimBX in %r27
        ld [dimBY], %r28        ! Laad dimBY in %r28


        ! Check de dimensies (aantal rijen van B = aantal kolommen van A), is voorwaarde om matrixvermenigvuldiging te kùnnen doen
        ! Trek dimBY (rijen in B, in %r28) van dimAX (kolommen in A, in %r28) af, indien gelijk, hebben we 0, indien niet, iets anders opgeslagen in %r3
        sub %r28, %r25, %r3
        andcc %r3, %r3, %r0     ! Conditiecode zetten, 0 indien GOED, iets anders indien dimensies niet overeenkomen
        bne DimError            ! Indien NIET gelijk aan 0, ga dan naar DimError, anders: ga verder (dimensies zijn oké)


        ! Nu beginnen we met het echte uitrekenen van de matrixvermenigvuldiging
        ! We implementeren de matrixvermenigvuldiging als volgt (pseudocode):
        !
        ! FOR i := 0 TO dimAY - 1 DO
        !       FOR j := 0 TO dimBX - 1 DO
        !               FOR k := 0 TO dimBY - 1 DO
        !                       C[i,j] := C[i,j] + A[i,k] * B[k,j]
        !               END;
        !       END;
        ! END;
```

```
                !
                ! We hebben dus 3 lussen die lopen en die de matrix C berekenen.
                ! Ik heb eerst bovenstaande geschreven in programma code en gechecked of dit correct was
                ! Vervolgens is dat ongeveer hetzelfde geïmplementeerd geweest, weliswaar in assembler en dus wel net iets anders...
                ! De drie FOR lussen van hierboven komen overeen met loop1, loop2 en loop3
                !
                ! Het principe is echter omgedraaid uitrekenen.  In plaats van zoals hierboven in de FOR lus van 0 tot dimAY-1 te gaan, gaan we van dimAY-1 tot 0
                ! Dit principe wordt genomen omdat het efficiënter is om te checken of %r3 op 0 staat (via 'be') dan te checken of we aan dimAY-1 zijn gekomen.
                ! We bouwen onze uitkomstenmatrix dus van rechtsonderaan tot linksbovenaan op, net omgekeerd dan we het met de hand zouden uitrekenen.
                ! Dit komt echter op exact hetzelfde uit.
                add %r26, %r0, %r3              ! zelfde als ld [dimAY], %r3
loop1:          andcc %r3, %r3, %r0        ! loop while %r3 <> 0  -- per rij in A -- voor alle rijen in result dus
                be CalcDone                     ! Indien %r3 == 0, dan springen naar CalcDone, dan zijn we klaar met de matrixvermenigvuldiging
                add %r27, %r0, %r4              ! zelfde als ld [dimBX], %r4
        loop2:          andcc %r4, %r4, %r0             ! loop while %r4 <> 0 -- voor alle kolommen in result  -- per kolom in B
                        be ContLoop1                    ! Indien %r4 = 0, dan is loop2 klaar en gaan we terug verder met loop1, dus gaan we naar ContLoop1 (komt van Continue
Loop 1)
                        add %r25, %r0, %r5              ! zelfde als ld[dimAX], %r5
                loop3:          andcc %r5, %r5, %r0             ! loop while %r5 <> 0 -- per kolom in A
                                be ContLoop2                    ! Indien %r5 == 0, dan is loop 3 klaar en gaan we verder met loop 2, dus ContLoop2
                                sub %r5, 1, %r5         ! Trek van %r5 eentje af, is voor de lus (zoals in C++:  k--;)

                                ! Nu gaan we de eigenlijke berekeningen doen in het programma
                                ! Dus de berekening van C[i,j] = C[i,j] + A[i,k] * B[k,j] (zie Oberon programma hierboven)

                                ! haal A[i,k] op = (adres van A + (((%r3 - 1) * dimAX) + r%5))
                                ! aangezien de matrixen niet als array maar als een opeenvolging van cijfers opgeslagen zijn in het geheugen,
                                ! gaan we ervoor zorgen dat we het juiste cijfer ophalen.  Daarom is het voor A[i,k] 'ingewikkeld' om dit te doen
                                ! we hebben dus het getal van de i-de rij, k-de kolom nodig (A[i,k]).
                                ! --> we nemen de gewenste rij (%r3), trekken er hier één van af (omdat we tellen vanaf 0),
                                ! vervolgens vermenigvuldigen we dit getal maal de dimAX, dus het aantal kolommen van A, zo weten we hoeveel elementen we moeten verder
                                ! zo weten we hoeveel elementen we moeten verder springen om tot aan de gewenste rij %r3 te geraken in de matrix
                                ! vervolgens, tellen we er %r5 bij op, dat is de gewenste kolom.  met dit cijfer, kunnen we + het adres van A, het gewenste cijfer ophalen
                                ! omdat we in het geheugen per 4 bytes verder gaan per getal, moeten we het uitgekomen getal nog vermenigvuldigen met 4
                                sub %r3, 1, %r30                ! %r3 - 1
                                add %r25, %r0, %r31                     ! zelfde als ld[dimAX], %r31
                                call multiply                   ! ((%r3-1) * dimAX) en opslaan in %r29
                                add %r29, %r5, %r30                     ! %r29 + %r5 (de gewenste kolom)
                                add %r0, 4, %r31                ! in %r31 zetten we 4
                                call multiply                   ! het %r30'ste getal * 4 = totaal aantal plaatsen te beginnen vanaf A voor gewenste getal, oplossing in %r29
                                ld A+%r29, %r11                         ! laad het getal A[i,k] in %r11

                                ! haal B[k,j] op = (adres van B + (((%r5 - 1) * dimBX) + %r4))
                                ! dit is juist hetzelfde principe als bij het vorige, aangezien matrix B op zelfde principe is opgeslagen in het geheugen
                                ! dit keer hebben we echter het element B[k,j] nodig (dus element op k-de rij, j-de kolom)
```

```
! --> we nemen de gewenste rij (%r5), trekken er hier één vanaf omdat we vanaf 0 beginnen te tellen, vervolgens vermenigvuldigen we dit
! met dimBX, het aantal kolommen van B en we tellen er %r4 bij op voor de juiste kolom; vervolgens nog eens maal 4 voor juiste geheugenplaats
add %r5, %r0, %r30              ! %r5
add %r27, %r0, %r31             ! zelfde als ld[dimBX], %r31
call multiply                   ! ((%r5 - 1) * dimBX) en opslaan in %r29
add %r29, %r4, %r30             ! vorige, %r29 + %r4 (de gewenste kolom)
sub %r30, 1, %r30               ! -1 omdat we het moeten aftrekken, we tellen vanaf 0 hier
add %r0, 4, %r31        ! in %r31 zetten we 4 (om zodadelijk maal 4 te kunnen doen)
call multiply                   ! het %r30'ste getal * 4 = totaal aantal plaatsen te beginnen vanaf B voor gewenste getal, oplossing in %r29
ld B+%r29, %r12                 ! laad het getal B[k,j]

! nu hebben we beide getallen opgehaald waarmee we willen vermenigvuldigen
! nu berekenen we A[i,k] * B[k,j]
add %r11, %r0, %r30             ! zet A[i,k] (staat in %r11) in %r30
add %r12, %r0, %r31             ! zet B[k,j] (staat in %r12) in %r31
call multiply                   ! voor de vermenigvuldiging A[i,k] * B[k,j] uit, oplossing in %r29
add %r29, %r0, %r13             ! zet de oplossing in %r13 voor later gebruik

! we hebben nu de vermenigvuldiging berekend, nu moeten we nog het originele getal C[i,j] ophalen, om het voorgaande bij op te tellen
! Dus ophalen C[i,j], zelfde principe als bij het ophalen van het element uit A en uit B,
! maar dit keer het element op i-de rij, j-de kolom
! C[i,j] = (adres van C + (((%r3 - 1) * dimBX) + %r4))
sub %r3, 1, %r30        ! trek van %r3 één af en sla dit op in %r30, dit is de i-de rij, maar -1 omdat we hier tellen vanaf 0
add %r27, 0, %r31               ! zelfde als ld [dimBX], %r31
call multiply           ! dus (%r3 - 1) * dimBX berekenen en oplossing in %r29
add %r29, %r4, %r30             ! voorgaande + %r4 (j-de kolom) en de oplossing in %r30
add %r0, 4, %r31        ! zet 4 in %r31, om zodadelijk maal 4 te doen
call multiply           ! het %r30'ste getal * 4 = totaal aantal plaatsen vanaf C voor het gewenste getal op te halen
sub %r29, 4, %r16               ! trek er 4 van de oplossing af, dit is omdat we anders C+4 doen (element 0 vergeten we anders)
ld C+%r16, %r17                 ! in %r17 staat nu C[i,j] de element

! nu hebben we alles wat we nodig hebben, we kunnen nu de optelling doen
! van het reeds gevonden element C[i,j] + de vermenigvuldiging van A[i,k] en B[k,j]
add %r13, %r17, %r17    ! dus C[i,j] + (A[i,k] * B[k,j]) en sla de uitkomst op in %r17

! nu het totale resultaat terug opslaan in het geheugen (store)
st %r17, C+%r16                 ! resultaat (%r17) opslaan in C+%r16, we weten %r16 nog van eerst het element op te halen, dus opslaan in C[i,j]

ba loop3                ! begin de loop3 terug opnieuw...
ContLoop2:      sub %r4, 1, %r4         ! we vervolgen hier de loop2, we trekken van onze variabele %r4 eentje af
                ba loop2                ! begin terug loop2 opnieuw...
ContLoop1:      sub %r3, 1, %r3         ! we vervolgen hier de loop1, we trekken van onze variabele %r3 eentje af
                ba loop1                ! begin opnieuw met loop1....
```

```
                ! Op dit ogenblik is de matrixvermenigvuldiging KLAAR.  We komen in CalcDone enkel via loop1 die dan àlles heeft afgewerkt
                ! We gaan nu de uitkomst die we nu hebben berekend gaan checken met een opgegeven uitkomst, die staat in CSol
                ! We voeren de lus uit per element van de uitkomstmatrix C, dus zo checken we élk element.
                ! Het aantal elementen van C is gelijk aan het aantal rijen van A * aantal kolommen van B
                ! Onderstaande assembler code kan in een meer visuele vorm geschreven worden in pseudocode als volgt:
                !
                ! AantalElementen = (dimAY * dimBX);
                ! FOR i := 0 TO AantalElementen DO
                !       j := i * 4;
                !       elem1 = C+j;
                !       elem2 = CSol+j;
                !       k = elem1 - elem2;
                !       if (k <> 0) then WrongSolution
                !       i--;
                ! END;
                !
                ! We doen ook dit weer van achter naar voren, van het laatste element tot het eerste element.  Dit heeft twee redenen.
                ! Ook hier is het efficiënter te controleren of onze variabele = 0 en niet gelijk is aan een bepaald getal AantalElementen
                ! Maar aangezien we de oplossingenmatrix ook van achter naar voren hebben berekend, is het het slimste deze ook van achteren naar voren te checken
                ! indien er iets fout gelopen is, is er zo waarschijnlijk minder werk om te checken.
CalcDone:       add %r0, 1, %r1         ! We zetten in %r1 de waarde '1' van oké.  Indien we hier niét meer aankomen, wil dit zeggen dat beide uitkomsten overeen komen
                add %r0, 4000, %r2              ! Er wordt ook gevraagd in %r2 de waarde van de geheugenlocatie te zetten waar de oplossing staat, dus C
                add %r26, %r0, %r30             ! zelfde als ld [dimAY], %r30
                add %r27, %r0, %r31             ! zelfde als ld [dimBX], %r31
                call multiply                   ! we vermenigvuldigen dimAY*dimBX, we doen dit om het totaal aantal elementen van onze uitkomst C te weten, uitkomst in %r29
                add %r29, %r0, %r3              ! we plaatsen de uitkomst in %r3
LoopCheck:      andcc %r3, %r3, %r0             ! while %r3 <> 0 -- we doen deze loop voor elk element in onze uitkomst, zo checken we élk element
                be Done                 ! indien %r3 == 0, dan zijn we klaar en kunnen we gaan naar Done
                add %r3, %r0, %r30              ! We nemen onze variabele van onze lus, %r2 en steken die in %r30
                add %r0, 4, %r31        ! we zetten 4 in %r31, om zodadelijk maal 4 te kunnen doen
                call multiply                   ! %r3 * 4 = aantal geheugenplaatsjes we verder gaan moeten gaan vanaf C en vanaf CSol voor het juiste element
                sub %r29, 4, %r4                ! we slaan de vorige uitkomst op in %r4 om terug te kunnen gebruiken, min 4 omdat we anders element 0 zouden vergeten
                ld C+%r4, %r5                   ! We laden het C+%r4'de element in %r5 (elem1 = C+j in bovenstaande pseudocode)
                ld CSol+%r4, %r6                ! We laden het CSol+%r4'de element in %r6 (elem2 = CSol+j in pseudocode)
                sub %r5, %r6, %r5               ! We trekken beide getallen van elkaar af
                andcc %r5, %r5, %r6             ! We vergelijken het en zetten een conditiecode; indien we ongelijke getallen hadden, komt er een 1, anders een 0
                bne WrongSolution               ! Indien beide getallen GELIJK zijn, dan doen we niets, indien ze NIET gelijk zijn, dan gaan we naar WrongSolution
                sub %r3, 1, %r3         ! Onze lusvariabele - 1 doen
                ba LoopCheck                    ! Start de checklus terug, dus terug naar LoopCheck verder gaan...

WrongSolution: add %r0, %r0, %r1                ! We komen hier indien er een verschil is gevonden tussen CSol en de berekende matrix C
                call Done                       ! Vorige regel: het register 1 vullen met 0, wat volgens afspraak aangeeft dat de matrix C en CSol niet overeen komen, ga dan naar
Done

Done:           halt                            ! Hier komen we indien heel het programma afgelopen is.  We kunnen hier zowel via LoopCheck komen, als via WrongSolution
```

```
            ! Aangezien we geen multiply hebben en enkel add en substract, moeten we de vermenigvuldiging maken, ook het werken met macro's gaat niet
            ! Voor multiply: A * B = C --> %r30 = A; %r31 = B; %r29 = C
            ! Voor we hier in komen, moet er worden gezorgd dat A in register %r30 zit, en B in %r31
            ! Vervolgens gaan we de vermenigvuldiging uitrekenen en zetten we de uitkomst in %r29.
            ! Deze drie registers zijn exclusief gereserveerd voor de vermenigvuldiging
            ! Het principe van vermenigvuldiging met enkel optellen en aftrekken:
            ! A * B = A + A + A + A + ... in totaal B keer
            ! Het wordt dus als lus opgevat.
            ! De pseudocode die het idee weergeeft:
            !
            ! i := B;
            ! WHILE i <> 0 DO
            !       C = C + A;
            !       i--;
            ! END;
            !
            ! We komen binnen op multiply, we intialiseren register 29, vervolgens gaat de lus enkel multiply2 uitvoeren.
multiply:   and %r0, %r0, %r29          ! in register 29 komt de uitkomst, dus leegmaken (dit zorgt ervoor indien vermenigvuldiging maal 0, dit ook goed werkt)
multiply2:  andcc %r31, %r31, %r0   ! while %r31 <> 0 blijf dit dan doen, dus B keer
            be MultDone                 ! Indien %r31 is nul, dan zijn we klaar
            add %r29, %r30, %r29        ! tel bij de gevonden oplossing een keer A bij op
            sub %r31, 1, %r31           ! trek van B (%r31) één af, dan staat in %r31 nog het aantal keren om te vermenigvuldigen
            ba multiply2                ! begin terug met multiply2
MultDone:   jmpl %r15 + 4, %r0          ! terug naar vorige positie gaan (dankzij de call, is de vorige positie in %r15 opgeslagen)

            ! Indien de dimensies fout zijn, komen we hier terecht: we kunnen de matrixen niet vermenigvuldigen, dus... we stoppen.
DimError:   halt

            .end                        ! Einde assembler code

! Hieronder staan de variabelen
dimAX: 2    ! kolommen in A
dimAY: 5    ! rijen in A
dimBX: 3    ! kolommen in B
dimBY: 2    ! rijen in B
A:     1
       2
       3
       5
       7
       11
       13
       17
       19
```

	23
B:	4
	6
	10
	8
	9
	12
CSol:	20
	24
	34
	52
	63
	90
	116
	141
	202
	188
	231
	334
	260
	321
	466